CS 433 Midterm Exam – March 7, 2016
Professor Sarita Adve

Time: 2 Hours

Please print your name and NetID and circle the appropriate category in the space provided below.

<table>
<thead>
<tr>
<th>Name</th>
<th>NetID</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td>Undergraduate</td>
</tr>
<tr>
<td></td>
<td></td>
<td>Graduate</td>
</tr>
</tbody>
</table>

Instructions

1. No books, papers, notes, or any other typed or written materials are allowed. No calculators or other electronic materials are allowed.

2. Please do not turn in loose scrap paper. Limit your answers to the space provided if possible. If this is not possible, please write on the back of the same sheet. You may use the back of each sheet for scratch work.

3. In all cases, show your work. No credit will be given if there is no indication of how the answer was derived. Partial credit will be given even if your final solution is incorrect, provided you show the intermediate steps in reaching the final solution.

4. If you believe a problem is incorrectly or incompletely specified, make a reasonable assumption and solve the problem. The assumption should not result in a trivial solution. In all cases, clearly state any assumptions that you make in your answers.

5. This exam has 6 problems and 10 pages (including this one). All students should solve problems 1 through 5B. Only graduate students should solve problems 5C and 6. Please budget your time appropriately. Good luck!

<table>
<thead>
<tr>
<th>Problem</th>
<th>Maximum Points</th>
<th>Received Points</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>16</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>18</td>
<td></td>
</tr>
<tr>
<td>5</td>
<td>6 (undergrads), 14 (grads)</td>
<td></td>
</tr>
<tr>
<td>6</td>
<td>6 (for graduates only)</td>
<td></td>
</tr>
<tr>
<td>Total</td>
<td>54 for undergraduates</td>
<td>68 for graduates</td>
</tr>
</tbody>
</table>
Problem 1 [4 points]
Assume 90% of a program can be executed in parallel.

Part A [2 points]
What speedup is required on the parallel section to achieve an overall speedup of 4X for the full program?
Solution: Let S be the speedup on the parallel section and T be the original total time.
.9T/S + .1T = T/4
.9/S + .1 = .25
.9/S = .15
S = .9 / .15 = 6. 6X speedup is required on the parallel section.
Grading: 2 points for the correct formula. 0 points for the final answer without showing the work.

Part B [2 points]
What is the maximum possible speedup achievable on the above program through parallelization?
Solution: In the limit, the parallel section reduces to 0, so speedup = T / 0.1T = 10X.
Grading: 2 points for the correct formula.
Problem 2 [10 points]

Consider branch prediction for the following loop. At the start of each iteration, a load reads a value that we refer to as the **input** to the loop. The first branch (B1) distinguishes the input as small (< 10) or big. Branch B2 tests whether a small number is even or odd. Branch B3 tests whether a big number is even or odd. B1 executes in each iteration, but only one of B2 or B3 execute in a given iteration. The jumps back to the top of the loop are unconditional, so they don’t affect the state or history of a branch predictor.

```
loop:
    LD R3, 0(R1) // The value loaded here is referred to as input below
    DADDI R1, R1, $8
    DADDI R4, R3, $-10
B1:  BGEZ R4, big // branch if input ≥ 10
small:
    ANDI R3, R3, $1
B2:  BNEZ R3, odd // branch if input < 10 and odd
even:
    DADDI R6, R6, $1
    J loop
odd:
    DADDI R7, R7, $1
    J loop
big:
    ANDI R3, R3, $1
B3:  BNEZ R3, odd-big // branch if input ≥ 10 and odd
even-big:
    DADDI R8, R8, $1
    J loop
odd-big:
    DADDI R9, R9, $1
    J loop
```

As an example, the table below gives the branch directions when the input seen is 8, 9, 12, 11, 8, 9, 12, 11, … Below, we refer to such an input as a “**repeating sequence of four inputs that is a permutation of the numbers 8, 9, 11, 12**.”

<table>
<thead>
<tr>
<th>iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>B1</td>
<td>N</td>
<td>N</td>
<td>12</td>
<td>11</td>
<td>8</td>
<td>9</td>
<td>12</td>
<td>11</td>
</tr>
<tr>
<td>B2</td>
<td>N</td>
<td>T</td>
<td></td>
<td></td>
<td>N</td>
<td>T</td>
<td></td>
<td></td>
</tr>
<tr>
<td>B3</td>
<td></td>
<td></td>
<td>N</td>
<td>T</td>
<td></td>
<td></td>
<td>N</td>
<td>T</td>
</tr>
</tbody>
</table>
The code from the previous page is repeated here for reference:

```assembly
loop:
    LD R3, 0(R1)                    // The value loaded here is referred to as input below
    DADDI R1, R1, $8
    DADDI R4, R3, $-10
B1:  BGEZ R4, big                  // branch if input ≥ 10
small:
    ANDI R3, R3, $1
B2:  BNEZ R3, odd                  // branch if input < 10 and odd
even:
    DADDI R6, R6, $1
    J loop
odd:
    DADDI R7, R7, $1
    J loop
big:
    ANDI R3, R3, $1
B3:  BNEZ R3, odd-big              // branch if input ≥ 10 and odd
even-big:
    DADDI R8, R8, $1
    J loop
odd-big:
    DADDI R9, R9, $1
    J loop
```

**Part A [3 points]** Give a repeating sequence of 4 inputs that is a permutation of the numbers 8, 9, 10, 11 such that a local (1, 1) correlating predictor for each branch will eventually predict every branch correctly. Use the following table to show the direction of the different branches with your sequence. Explain your answer.

**Solution:** For a local (1, 1) predictor to be completely accurate on a branch, it must alternate or always take the same direction. A permutation that works is 8 11 9 10 as follows:

<table>
<thead>
<tr>
<th>iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>8</td>
<td>11</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>11</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>B1</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>B2</td>
<td>N</td>
<td>-</td>
<td>T</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>B3</td>
<td>-</td>
<td>T</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>T</td>
<td>-</td>
<td>N</td>
</tr>
</tbody>
</table>

**Grading:** 3 points for a correct answer with a reasonable explanation. For partial credit: 1 point for describing intuitively when a local (1, 1) predictor works. 1 point for filling in the table above correctly for a reasonable sequence. 1 point for giving the right permutations.

Deductions: -½ point if permutation is not from given numbers. -½ point if explanation lacks sufficient detail.
Part B [3 points] Give a repeating sequence of 4 different input values such that a global (1, 1) correlating predictor for branches B2 and B3 will eventually predict every invocation of these branches correctly. Your input sequence should be such that B1 changes direction at least twice. Use the following table to show the direction of the different branches with your sequence. Explain your answer.

Solution: The 1 bit global history at B2 is always not taken, because B2 is only reached on an iteration if B1 was not taken. For the global (1, 1) predictor to be always correct, B2 must always go the same direction. The same goes for B3. An input sequence like 8, 11, 6, 13, 8, 11, 6, 13, works (it also changes the direction of B1) as shown below.

<table>
<thead>
<tr>
<th>iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input</td>
<td>8</td>
<td>11</td>
<td>6</td>
<td>13</td>
<td>8</td>
<td>11</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>B1</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>B2</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>N</td>
<td>-</td>
</tr>
<tr>
<td>B3</td>
<td>-</td>
<td>T</td>
<td>-</td>
<td>T</td>
<td>-</td>
<td>T</td>
<td>-</td>
<td>T</td>
</tr>
</tbody>
</table>

Grading: 3 points for a correct answer with a reasonable explanation. 1 point partial credit for a sequence where the global predictor is always correct, but B1 never changes direction. 1 point partial credit for filling in the above table correctly for a reasonable sequence. 1 point partial credit for describing intuitively when a global predictor works.

Deductions: -1 for no/wrong explanation; -½ if explanation lacks sufficient detail.

Part C [4 points] Give a repeating sequence of 4 different input values such that a local (1, 1) correlating predictor for each branch will eventually predict every branch correctly, but a global (1, 1) predictor for each branch will continue to make some errors for at least one of the three branches. Use the following table to show the direction of the different branches with your sequence. Explain your answer.

Solution: For the local predictor to be completely accurate, each branch must either always take the same direction or alternate. To force an error in the global (1, 1) predictor, it is enough that B2 or B3 change direction. The input 8 11 9 10 8 11 9 10 … works as follows:

<table>
<thead>
<tr>
<th>iteration</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>7</th>
<th>8</th>
</tr>
</thead>
<tbody>
<tr>
<td>input</td>
<td>8</td>
<td>11</td>
<td>9</td>
<td>10</td>
<td>8</td>
<td>11</td>
<td>9</td>
<td>10</td>
</tr>
<tr>
<td>B1</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
<td>N</td>
<td>T</td>
</tr>
<tr>
<td>B2</td>
<td>N</td>
<td>-</td>
<td>T</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>T</td>
<td>-</td>
</tr>
<tr>
<td>B3</td>
<td>-</td>
<td>T</td>
<td>-</td>
<td>N</td>
<td>-</td>
<td>T</td>
<td>-</td>
<td>N</td>
</tr>
</tbody>
</table>

Grading: 2 points if a local (1, 1) predictor will make no errors. 2 points if a global (1, 1) predictor will continue to make errors.

Deductions: -1 for no/wrong explanation; -½ if explanation lacks sufficient detail. -1 for wrong table entries.
Problem 3 [16 points]

This problem concerns Tomasulo’s algorithm (with reservation stations) with the reorder buffer as discussed in detail in the lecture notes, with the following changes/additions/clarifications.

- Assume only the following functional units:

<table>
<thead>
<tr>
<th>Functional Unit</th>
<th>Cycles in EX</th>
</tr>
</thead>
<tbody>
<tr>
<td>Integer Add</td>
<td>1</td>
</tr>
<tr>
<td>Integer Mul</td>
<td>3</td>
</tr>
<tr>
<td>Integer Div</td>
<td>7</td>
</tr>
</tbody>
</table>

- The processor can issue and commit at most one instruction per cycle.
- The CDB supports only one broadcast per cycle.
- Assume you have an unlimited number of reservation stations and reorder buffer entries.
- An instruction waiting for data on the CDB can move to EX in the cycle after the CDB broadcast.
- An instruction waiting to write to the CDB holds its execution unit until it gets the CDB; i.e., it prevents other instructions needing the same functional unit from beginning execution.
- Assume that integer instructions also follow Tomasulo’s algorithm (analogous to the floating point instructions) so they can be issued out of order and the result from the integer functional unit is also broadcast on the CDB and forwarded to dependent instructions through the CDB.

Complete the blank entries in the following table using the above specifications. For each instruction, fill in the cycle numbers in each pipeline stage (CM stands for commit) and indicate where its source operands are read from (use RF for register file, ROB for reorder buffer, and CDB for common data bus). The entries for the first instruction and for the issue stage are filled in for you (Op means operand). Entries with dots should be ignored. Assume that at the start of the code snippet below, all registers have up-to-date values.

<table>
<thead>
<tr>
<th>IS</th>
<th>Op1</th>
<th>Op1 source</th>
<th>Op2</th>
<th>Op2 source</th>
<th>EX</th>
<th>WB</th>
<th>CM</th>
</tr>
</thead>
<tbody>
<tr>
<td>MUL R2, R6, R12</td>
<td>1</td>
<td>R6</td>
<td>RF</td>
<td>R12</td>
<td>RF</td>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>ADD R1, R1, R2</td>
<td>2</td>
<td>R1</td>
<td>RF</td>
<td>R2</td>
<td>CDB</td>
<td>6</td>
<td>7</td>
</tr>
<tr>
<td>DIV R2, R2, R4</td>
<td>3</td>
<td>R2</td>
<td>CDB</td>
<td>R4</td>
<td>RF</td>
<td>6</td>
<td>13</td>
</tr>
<tr>
<td>ADDI R7, R4, #4</td>
<td>4</td>
<td>R4</td>
<td>RF</td>
<td>...</td>
<td>...</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>ADD R5, R5, R7</td>
<td>5</td>
<td>R5</td>
<td>RF</td>
<td>R7</td>
<td>CDB</td>
<td>7</td>
<td>8</td>
</tr>
<tr>
<td>MUL R6, R3, R5</td>
<td>6</td>
<td>R3</td>
<td>RF</td>
<td>R5</td>
<td>CDB</td>
<td>9</td>
<td>12</td>
</tr>
<tr>
<td>ADDI R9, R7, #8</td>
<td>7</td>
<td>R7</td>
<td>ROB</td>
<td>...</td>
<td>...</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>ADD R5, R5, R6</td>
<td>8</td>
<td>R5</td>
<td>ROB</td>
<td>R6</td>
<td>CDB</td>
<td>13</td>
<td>14</td>
</tr>
</tbody>
</table>

**Grading:** 1/2 pt each entry (32 total entries). No deduction for entries which are correct given earlier entries, but repeat errors will be penalized.
Problem 4 [18 points]

Consider the following code fragment:

Loop:
- LD.D F1, 0(R1)
- LD.D F2, 0(R2)
- MUL.D F3, F1, F10
- ADD.D F4, F3, F2
- SD.D F4, 0(R1)
- DADDUI R1, R1, #8
- DADDUI R2, R2, #8
- BNE R1, R3, Loop

Consider a pipeline with the following latencies: 3 cycles between an FP multiply and its consumer, 1 cycle between an FP add and its consumer, and 0 cycles between all other pairs. Thus, there should be three stall cycles between the multiply and addition in the above code for correct operation.

Unroll the above loop 4 times and write the resulting code to the left of the table below. You have access to temporary registers T0…T63. Assume that the total number of iterations for the original loop is a multiple of 4. Then, schedule the unrolled loop for a VLIW machine where each VLIW instruction can contain one memory reference, one FP operation, and one integer operation. Write the scheduled instructions in the table below to minimize the number of stalls. You may use L for L.D, M for MUL.D, etc.

This solution assumes pipelined functional units. Solutions with non-pipelined functional units were also accepted.

<table>
<thead>
<tr>
<th>Mem</th>
<th>FP ALU</th>
<th>Integer ALU</th>
</tr>
</thead>
<tbody>
<tr>
<td>LD.D F1, 0(R1)</td>
<td>MUL.D F3, F1, F10</td>
<td>DADDUI R1, R1, #32</td>
</tr>
<tr>
<td>LD.D F2, 0(R2)</td>
<td>MUL.D T3, T1, F10</td>
<td>DADDUI R2, R2, #32</td>
</tr>
<tr>
<td>LD.D T1, 8(R1)</td>
<td>ADD.D F4, F3, F2</td>
<td></td>
</tr>
<tr>
<td>LD.D T2, 8(R2)</td>
<td>ADD.D T4, T3, T2</td>
<td></td>
</tr>
<tr>
<td>LD.D T3, 16(R1)</td>
<td>ADD.D T5, 24(R1)</td>
<td>ADD.D T8, T7, T6</td>
</tr>
<tr>
<td>LD.D T4, 24(R1)</td>
<td>ADD.D T6, 16(R2)</td>
<td>ADD.D T10, 24(R2)</td>
</tr>
<tr>
<td>LD.D T5, 24(R1)</td>
<td>ADD.D T7, T5, F10</td>
<td></td>
</tr>
<tr>
<td>LD.D T6, 16(R2)</td>
<td>ADD.D T8, T7, T6</td>
<td></td>
</tr>
<tr>
<td>SD.D T9, 24(R1)</td>
<td>ADD.D T10, 24(R2)</td>
<td></td>
</tr>
<tr>
<td>SD.D T11, T9, F10</td>
<td>ADD.D T11, T10</td>
<td></td>
</tr>
<tr>
<td>ADD.D T12, T11, T10</td>
<td>SD.D T12, T12, T12</td>
<td></td>
</tr>
<tr>
<td>SD.D T12, 24(R1)</td>
<td>SD.D T12, T12, T12</td>
<td></td>
</tr>
<tr>
<td>DADDUI R1, R1, #32</td>
<td>BNE R1, R3, Loop</td>
<td></td>
</tr>
<tr>
<td>DADDUI R2, R2, #32</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BNE R1, R3, Loop</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
**Grading:** 9 points for code on left, 9 points for table. Minimum score is 0 for both.

Deductions (only for first error, no cascading errors):

- -2 points for badly ordered loads.
- -2 points for inconsistent load/store addresses.
- -2 points for badly ordered MULs or ADDs.
- -2 points for late stores.
- -2 points for late branches/integer operations.
- -2 points for missing or wrong-path operations.
- -2 points if instruction uses wrong functional unit.
- -2 points if R1/R2 are updated every iteration.
- -2 points for inconsistent DADDUI offsets.
- -2 points for violating a RAW dependency.
- -2 points for not using temporary registers.
- -1 point for using wrong registers (F instead of T, or vice versa).
- -1 point if temporary registers are out of bounds (not from T0-T63).
- -1 point if branch delay slot used but not stated in assumptions.

If you did not write the unrolled code, we immediately deducted ½ point. Then we took your VLIW scheduled code and graded it according to the above deductions.
Problem 5 [14 points]

Assume an in-order single-issue pipeline with multiple functional units. IF, ID, and WB take 1 cycle as usual, branches are resolved in ID, and all needed forwarding paths are provided. There is a fully pipelined, 3 cycle FP add unit (i.e., 3 cycles elapse from the time an FP add enters its functional unit to the time its result is available for a later instruction). There is a fully pipelined 5 cycle FP multiplier, and a 1 cycle integer ALU for all non-branch integer instructions. Loads and stores execute for 1 cycle in the integer ALU and spend 12 cycles in MEM. Consider the following code on the above machine:

```
loop:
  LD.D   F4, 0(R1)
  LD     R2, 4096(R1)
  LD.D   F10, 0(R2)
  MUL.D  F6, F4, F10
  ADD.D  F8, F8, F6
  DADDI  R1, R1, #8
  BNEQ   R1, R3, loop  // branch if R1 does not equal R3
```

Part A [4 points]

Give one pair of instructions with a RAW dependence that requires the most separation between them to avoid stalling? What is the minimum number of instructions that would need to be between this pair in program order to avoid the stall? Explain your answer.

Solution: A load and an instruction that uses the load’s result require the most separation. Any such pair is acceptable. Each of these pairs requires 12 instructions of separation. The load produces a result after 1 cycle of EX and 12 of MEM, and in this code all the instructions that need the result of the load will use that result in EX.

Grading: 2 points for any correct pair with correct explanation. 2 points for the number of instructions needed for separation. 1 point partial credit on the separation if the correct forwarding (MEM->EX) is mentioned.

Part (B) [2 points]

Consider the two dependent instructions from Part A. Just for this part, ignore all other dependences in the loop body; i.e., assume that dependences for all other instructions magically do not require any stalls. Now consider unrolling the loop to remove any stalls between the instructions chosen in Part (A). What is the minimum number of original iterations that the new unrolled loop would need to have to eliminate these stalls? Remember to explain your answer.

Solution: From part (a), a loop must be at least 14 instructions long to avoid stalling. There are 5 instructions of the loop body which will be multiplied by unrolling, and two instructions of loop overhead which will not. Having two of the original iterations in the unrolled loop is insufficient (2*5+2 = 12 < 14), but 3 iterations is enough.

Grading: 2 points for correct count, but only with explanation. 1 point partial credit for finding the number of instructions needed in the unrolled loop.
Part C – ONLY GRADUATE STUDENTS SHOULD SOLVE THIS PROBLEM [8 points]

Now consider the original loop with all the original dependences and stalls as in Part A. Software pipeline the loop to run in as few cycles as possible, using the start-up code given below. You only have to produce the code for the steady state. Do not write or consider the performance of the clean-up code. Do not modify the start-up code. Do not unroll the loop.

Startup code:

```
LD  R2, 4096(R1) // iteration 1
LD.D F4, 0(R1)  // iteration 1
LD.D F10, 0(R2) // iteration 1
LD  R2, (8+4096)(R1) // iteration 2
DADDI R1, R1, #16 // starts iteration 3
```

Your steady-state code here:

Solution:

```
loop:

MUL.D F6, F4, F10 // iteration x-2
LD.D F4, -8(R1) // iteration x-1
LD.D F10, 0(R2) // iteration x-1
LD  R2, 4096(R1) // iteration x
ADD.D F8, F8, F6 // iteration x-2
DADDI R1, R1, #8
BNEQ R1, R3, loop
```

This avoids using load results for as long as possible. 8 stall cycles still remain because the multiply depends on both loads.

Grading: Deduct 1/4 point for each extra stall cycle. Deduct ½ point for loads with incorrect offsets. Deduct 1 point for other incorrect instructions.
Problem 6 - ONLY GRADUATE STUDENTS SHOULD SOLVE THIS PROBLEM [6 points]

You are a member of a team designing an out-of-order processor with dynamic scheduling and speculative execution. Your initial design was just reviewed by the circuit implementation team, and it turns out that you have some spare transistor budget! (A rare occurrence in practice.)

Your processor currently has a small 2-bit saturating counter-based branch predictor which performs moderately well. It has 8 Integer Functional Units and 4 Floating Point Units (FPUs), 256KB of on-chip caches, 4 reservation stations for the Integer Units, and 2 reservation stations for FPUs. The Reorder Buffer has 8 entries. The processor has a 25 stage pipeline.

The applications you care for have a small code size and work on small data sets in the range of 64 KB. These applications spend most of their time in loops whose iterations are independent of each other, but typically have only a limited amount of ILP within a single iteration (within the current processor implementation).

You can use the extra transistors in (possibly several of) the following ways:
1. Improve the branch predictor accuracy.
2. Add more reservation stations to your Tomasulo’s Algorithm-based Dynamic Scheduler.
3. Add more FPUs and Integer Units.
4. Add more Reorder Buffer entries.

Some of these may be desirable additions, while others may not be too beneficial given the current configuration. There is a meeting coming up to discuss the proposed additions. Which of the above four additions should you support and which ones should you oppose (you can support/oppose multiple of these)? You need to justify your choices to receive credit.

Solution:

1. Improve the branch predictor accuracy: This is a desirable addition. The problem states that the branch predictor performs only moderately well and the processor has a long pipeline making branch mispredictions expensive. Thus, improving branch prediction accuracy is quite likely to increase performance.

2. Add more reservation stations to your Tomasulo’s Algorithm-based Dynamic Scheduler: This is a desirable addition. More reservation stations would mean a larger window within which the processor can search for ready instructions to execute, thus it can discover more parallelism and keep execution units busy. This would lead to better performance, especially since our application needs to discover parallelism across loop iterations.

3. Add more FPUs and Integer Units: This doesn’t seem to be a good addition. The current machine already has enough FUs and we should try to improve other aspects of the processor. Adding more FUs won’t help if the processor is unable to discover enough parallelism in the instruction stream to keep them busy.

4. Add more Reorder Buffer entries: This is a desirable addition. The current configuration has very few ROB entries. A large ROB helps to mask out the effects of long latency instructions and help search for parallelism within a larger window (this goes together with (2)).

Grading: 1.5 points for correctly analyzing each part. 6 points total. Give partial credit (0.5 points) if student gives valid reason for why improvement can be avoided.